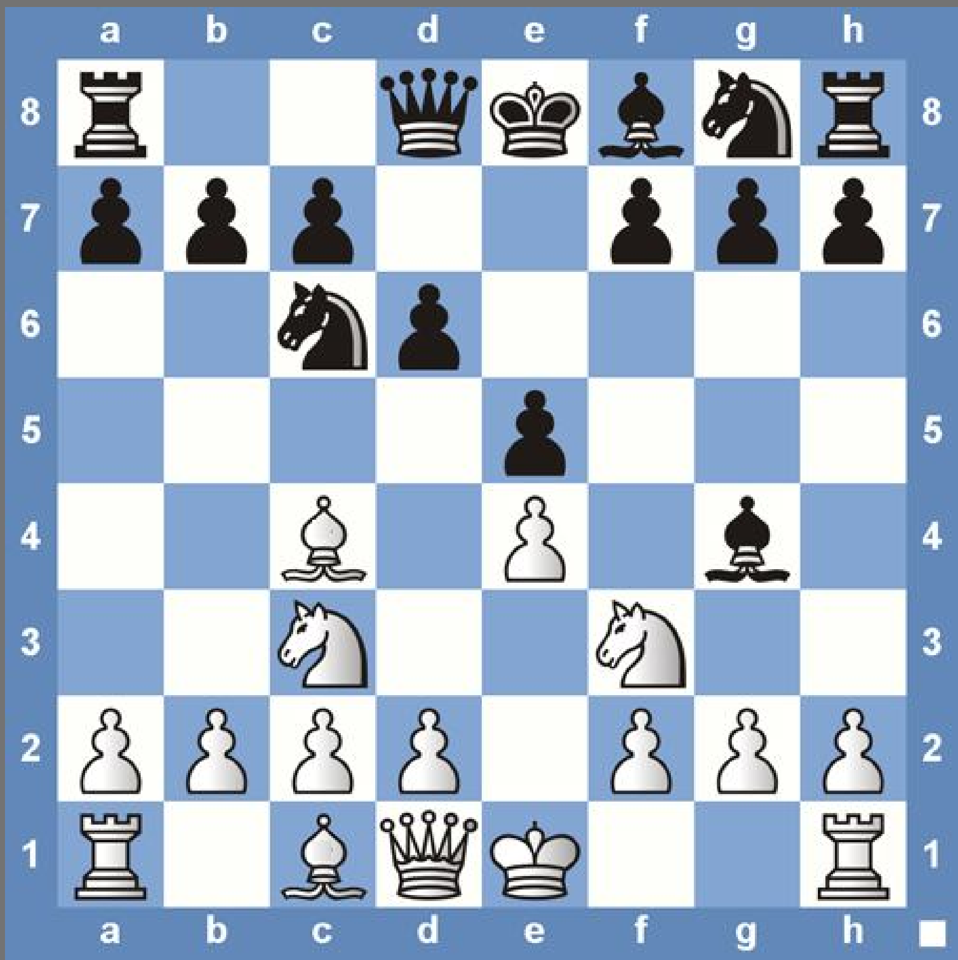
2012

تعرف على الذكاء الصناعي

إجعل الحاسوب يفكر و يلعب الشرطرنج



س

ياسين الجزائري

khatibe_30@hotmail.fr



الحمد لله رب العالمين والصلاة و السلام على المبعوث الأمين رحمة للعالمين محمد بن عبد الله و على آله و صحبه و سلم تسليما كثيرا.

لا يمكن نشر أي لعبة ذات لاعبين و توزيعها دون إضافة خاصية اللعب ضد الكومبيوتر, قد يبدو الموضوع معقدا للوهلة الأولى و تطبيقه صعب جدا إذ أنه يعتبر ذكاءا صناعيا, لكن على العكس, الأمر و تطبيقه ليس أصعب من برمجة اللعبة في حد ذاتها طبعا بعد التعرف على الألغوريتم المناسب و تعلم برمجته, و لهذا أقدم لكم هذا الكتيب آملا أن يوضح لكم كيف تتم العملية.

من الضروري جدا أن تحيط علما بأساسيات لغة البرمجة C++ لأنها ما سنعتمد عليه في هذا الكتاب بالإضافة إلى معرفة و إن كانت سطحية حول إستعمال المكتبة OpenGL الرسومية.

الفهرس

مقدمة.

الألغوريتم minimax يتم تطبيقه على الألعاب ذات لاعبين, مثل تيك تاك تو, الشطرنج, الداما و غيرهم, هذه الألعاب التي يمكن تطبيق الألغوريتم عليها تشترك في خاصيتين:

1. الألعاب ذات 'مجموع نتائج اللاعبين' يساوي الصفر, مثلا في لعبة الشطرنج, إذا افترضنا أن الفائز يأخذ النتيجة 1, الخاسر يأخذ -1 و في حالة التعادل كل لاعب يأخذ 0, نجد أن مجموع نتائج اللاعبين مساوي للصفر.
2. الألعاب ذات 'معلومات كاملة', أي أن كل لاعب يعرف كل شيء عن كل الحركات الممكنة الي سيقوم بها اللاعب الآخر و نتائجها.

كما أنها ألعاب ذات قواعد معروفة و محددة, و بهذا يمكن في أي مرحلة من اللعبة معرفة الحركات الممكنة التالية.

الألغوريتم minimax له عدة تحسينات كـ negamax و alpha-beta, لذلك, سأقدم أولا شرحا مختصرا عن برمجة اللعبة التي سنطبق عليها الألغوريتم, لعبة Tic Tac Toe المشهورة, ثم نبدأ بالتعرف على كل ألغوريتم و نطبقه على اللعبة على حدا و نرى الفرق.

2. برمجة لعبة Tic Tac Toe

سنستعمل لغة السي ++ تحت بيئة التطوير Visual studio 2005 أو أي نسخة بعدها, و كمكتبة للرسوميات, نستعمل OpenGL مع المكتبة المساعدة OpenGlut.

لذلك, قم بتحميل كل من المكتبة OpenGL و OpenGlut و انسخ الملفات h.* إلى المجلد:

C:\Program Files\Microsoft Visual Studio 8\VC\include\GL

و الملفات lib.* إلى المجلد:

C:\Program Files\Microsoft Visual Studio 8\VC\lib

طبعا على افتراض أن الفيجوال ستديو مثبت على القرص المحلي C.

الخطوة الأخيرة في تثبيت المكتبة هي بنسخ الملف OpenGLUT.dll إلى مجلد النظام:

C:\WINDOWS

شرح مفصل و بالصور للأساسيات و تثبيت المكتبة موجود في دروس سابقة.

افتح الفيجوال ستديو و قم بإنشاء مشروع جديد و فارغ باسم TicTacToe, ثم أضف إليه ثلاث ملفات, main.cpp, game.h و game.cpp, ما سنبرمجه الأن هو لعبة تيك تاك تو بسيطة بين لاعبين.

سنمر مرورا سريعا على الكود لأن برمجة اللعبة في حد ذاتها ليس هدفنا, و إذا أردت شرحا مفصلا حول برمجة الرسوميات باستخدام المكتبة OpenGL قم بالإطلاع على الدروس السابقة.

الملف game.h سنعرف فيه الكلاس CGame التي ستقوم بتسيير اللعبة, افتح الملف game.h و اكتب فيه الكود التالي:

```
#pragma once
#include <GL/openglut.h>
#include <math.h>
#include <iostream>

static enum {O, X, EMPTY};
static enum {OWIN, XWIN, DRAW, PLAYING};

#define PI          3.14159265
```

```
class CGame
{
public:
	int m_clickedX;
	int m_clickedY;
	int m_turn;
private:
	int m_board[3][3];
	int  m_state;
	void resetGame(void);
	void showMessage(float shiftRight, char *message);
	void drawPiece(int piece, int row, int column);
	void drawBoard(void);
	int checkGameState(int forPlayer);
public:
	CGame(void);
	~CGame(void);
	void play(void);
};
```

إذا, أول ما تم تعريفه, قبل تعريف الكلاس CGame, هو الثوابت O, X, XWIN, ... ثوابت عادية نستعملها بدل أن نعطي لكل حالة أو لاعب رقم معين.

PI سنستخدمه في رسم الدوائر الخاصة باللاعب O.

الكلاس CGame, بها ثلاث متغيرات, m_clickedX و m_clickedY كل منهما سيأخذ إحداثيات الخانة التي تم عليها الكليك بغرض رسم دائرة أو اكس بها, حسب اللاعب الذي قام بالكليك.

m_turn سيأخذ إحدى القيمتين, إما O أو X, سيأخذ القيمة التي تمثل دور اللاعب الحالي, هل هو X أو O.

المصفوفة m_board هي مصفوفة اللعب ذات التسع خانات.

m_state سيأخذ أحد القيم XWIN, OWIN, DRAW أو PLAYING أو حالة اللعبة حاليا.

بقية الدوال الست, واضح من اسم كل دالة دورها, هذا الجدول يلخص دور كل دالة:

resetGame	تفريغ الخانات و إعادة اللعبة إلى حالتها الإبتدائية.
showMessage	إضهار رسالة نصية عادية على اللعبة.
drawPiece	رسم X أو O على مكان معين على الشاشة.
drawBoard	رسم لوحة اللعبة بكل الرموز اللتي تم وضعها حاليا.
checkGameState	التحقق من حالة اللعبة, فوز أحد اللاعبين أو التعادل أو

مواصلة اللعب	
الدالة التي تمرر الأدوار بين اللاعبين و تثبت حركة كل لاعب	play

الخطوة التالية, كتابة الكود الخاص بالملف game.cpp و تعريف جسم الكلاس CGame و تعريف دوالها, انسخ الكود التالي على الملف:

```
#include "Game.h"

CGame::CGame(void)
{
	this->resetGame();
}

CGame::~CGame(void)
{
}

void CGame::resetGame(void){
 this->m_board[0][0] = this->m_board[0][1] = this->m_board[0][2] =
 this->m_board[1][0] = this->m_board[1][1] = this->m_board[1][2] =
 this->m_board[2][0] = this->m_board[2][1] = this->m_board[2][2] =
EMPTY;
	this->m_clickedX = -1;
	this->m_clickedY = -1;
	this->m_state = PLAYING;
	this->m_turn = X;
}

void CGame::showMessage(float shiftRight, char *message)
{
  glPushMatrix();
  glColor3f(0, 1, 0);
  glTranslatef(-shiftRight, 0, 0);
  glScalef(0.003, 0.004, 0.004);
  while(*message){
    glutStrokeCharacter(GLUT_STROKE_ROMAN, *message);
    message++;
  }
  glPopMatrix();
}

void CGame::drawBoard(){
	glColor4f(1, 1, 1, 1);
	glBegin(GL_QUADS);
	 glVertex2f(-1.5, -1.5);glVertex2f(-1.5, 1.5);
	 glVertex2f(1.5, 1.5);glVertex2f(1.5, -1.5);
	glEnd();
	glColor4f(0, 0, 0, 0);
	glLineWidth(2);
```

```
     glBegin(GL_LINES);
      glVertex2f(-0.5, 1.5);glVertex2f(-0.5, -1.5);
      glVertex2f(0.5, 1.5);glVertex2f(0.5, -1.5);
      glVertex2f(-1.5, -0.5);glVertex2f(1.5, -0.5);
      glVertex2f(-1.5, 0.5);glVertex2f(1.5, 0.5);
     glEnd();
     glLineWidth(5);
     for(int i = 0; i < 3; i++)
          for(int j = 0; j < 3; j++)
               this->drawPiece(this->m_board[i][j], i, j);
     glLineWidth(2);
}
void CGame::drawPiece(int piece, int row, int column){
 if(piece == X){
  glColor4f(0, 0, 1, 1);
  glBegin(GL_LINES);
  glVertex2f(-1.4 + column, 1.4 - row);
  glVertex2f(-0.6 + column, 0.6 - row);
  glVertex2f(-1.4 + column, 0.6 - row);
  glVertex2f(-0.6 + column, 1.4 - row);
  glEnd();
 }else if(piece == O){
  glColor4f(1, 0, 0, 1);
  glBegin(GL_LINE_STRIP);
  for(int i = 0; i <= 20; i++)
  glVertex2f(sin((2*PI*i)/20)/3-1+column,cos((2*PI*i)/20)/3+1-row);
  glEnd();
 }
}

int CGame::checkGameState(int forPlayer) {
     if ((this->m_board[0][0] == forPlayer && this->m_board[0][1]
== forPlayer && this->m_board[0][2] == forPlayer) ||
          (this->m_board[1][0] == forPlayer && this->m_board[1][1]
== forPlayer && this->m_board[1][2] == forPlayer) ||
          (this->m_board[2][0] == forPlayer && this->m_board[2][1]
== forPlayer && this->m_board[2][2] == forPlayer) ||
          (this->m_board[0][0] == forPlayer && this->m_board[1][0]
== forPlayer && this->m_board[2][0] == forPlayer) ||
          (this->m_board[0][1] == forPlayer && this->m_board[1][1]
== forPlayer && this->m_board[2][1] == forPlayer) ||
          (this->m_board[0][2] == forPlayer && this->m_board[1][2]
== forPlayer && this->m_board[2][2] == forPlayer) ||
          (this->m_board[0][0] == forPlayer && this->m_board[1][1]
== forPlayer && this->m_board[2][2] == forPlayer) ||
          (this->m_board[0][2] == forPlayer && this->m_board[1][1]
== forPlayer && this->m_board[2][0] == forPlayer)){
               if(forPlayer == X) {
                    return XWIN;
               } else if(forPlayer == O) {
                    return OWIN;
```

```
				}
		}
		else {
			for(int i = 0; i < 3; i++)
				for(int j = 0; j < 3; j++)
				 if(this->m_board[i][j] == EMPTY)
					return PLAYING;
		}
		return DRAW;
}

void CGame::play(void){
 if(this->m_state == PLAYING)  {
  if(this->m_turn==O&&this->m_clickedX!=-1&&this->m_clickedY!= -1){
   if(this->m_board[this->m_clickedX][this->m_clickedY] == EMPTY){
    this->m_board[this->m_clickedX][this->m_clickedY] = O;
    this->m_turn = X;
    this->m_state = this->checkGameState(O);
   }
   this->m_clickedX = this->m_clickedY = -1;
 }else if(this->m_clickedX != -1 && this->m_clickedY != -1){
  if(this->m_board[this->m_clickedX][this->m_clickedY] == EMPTY){
   this->m_board[this->m_clickedX][this->m_clickedY] = X;
   this->m_turn = O;
   this->m_state = this->checkGameState(X);
 }
 this->m_clickedX = this->m_clickedY = -1;
 }
}
this->drawBoard();
if(this->m_state == XWIN ) this->showMessage(2.3, "X won!");
if(this->m_state == OWIN ) this->showMessage(2.3, "O won!");
else if(this->m_state == DRAW ) this->showMessage(0.5, "Draw.");
}
```

ما يهمنا شرحه في هذا الدرس هو الدالة play, أول ما سنتحقق منه عند إستدعاء هذه الدالة هو `m_state == PLAYING`, إي سنتحقق من أن اللعبة لم تنتهي.

ثم نريد أن نعرف لمن الدور حاليا, للاعب X أو اللاعب O, هذه المعلومة مخزنة في المتغير `m_turn` و الذي ستكون قيمته الإبتدائية X حسب الدالة `resetGame`.

بعد معرفة اللاعب, نتحقق من أن الإحداثيات الموجودة في `m_clickedX` و`m_clickedY` صحيحة, و هذا بكون قيمها غير مساوية لـ 1-.

طبعا يجب أن تكون الخانة التي تم الضغط عليها من طرف اللاعب على اللوحة فارغة, [أ أنه يجب أن تكون :

```
m_board[this->m_clickedX][this->m_clickedY] == EMPTY
```

إذا تحققت الشروط السابقة, هذا يعني أن الأمور سارت عى ما يرام و نضع الإشارة المناسبة على الخانة التي اختارها اللاعب ثم نمرر الدور للاعب الآخر و نتحقق من حالة اللعبة و إذا ما كان هناك فائز أم لا و نخرج من الدالة إلى أن يتم إستدعائها مرة أخرى.

بقي شيء واحد في هذه المرحلة, و هو كتابة محتوى الملف main.cpp, الملف الرئيسي في المشروع, و هذا الكود الي يجب أن يكتب فيه:

```
#define _WIN32_WINNT 0x0500
#define WINVER 0x0501
#include <windows.h>
#include <GL/openglut.h>
#include "Game.h"
#pragma comment(lib, "GLU32.LIB")

CGame *game;
void Reshape(int w, int h)
{
      glViewport(0, 0, w, h);
      glMatrixMode(GL_PROJECTION);
      glLoadIdentity();
      gluPerspective(45.0, (float)w/(float)h, 1.0, 100.0);
      gluLookAt(0,0,6,0,0,0,0,1,0);
      glMatrixMode(GL_MODELVIEW);
      glLoadIdentity();
      glutReshapeWindow(500, 500);
}

void Display(void)
{
      glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
      glLoadIdentity();
      game->play();
      glutSwapBuffers();
}
void Key(unsigned char key, int x, int y )
{
      if(key ==  27 )     exit(0);
}
void action(void)
{
      Display();
}
void mouse(int button,int state,int x,int y)
{
 switch(button)
 {
  case GLUT_LEFT_BUTTON:
      if(state==GLUT_DOWN){
```

```
            game->m_clickedX = (int)(y / 100) - 1;
            game->m_clickedY = (int)(x / 100) - 1;
            if(game->m_clickedX<0 || game->m_clickedX > 2 |
                 game->m_clickedY<0 || game->m_clickedY > 2){
                 game->m_clickedX = -1;
                 game->m_clickedY = -1;
            }
       }
 }
}
void main(int argc, char **argv)
{
     game = new CGame();
     HWND hWnd = GetConsoleWindow();
     ShowWindow( hWnd, SW_HIDE );
     glutInit(&argc, argv);
     glutInitDisplayMode(GLUT_RGBA | GLUT_DOUBLE);
     glutInitWindowSize(500, 500);
     glutInitWindowPosition(100,0);
     glutCreateWindow("TicTacToe");
     glEnable(GL_LINE_SMOOTH);
     glShadeModel(GL_SMOOTH);
     glHint(GL_LINE_SMOOTH_HINT, GL_DONT_CARE);
     glutDisplayFunc(Display);
     glutReshapeFunc(Reshape);
     glutKeyboardFunc(Key);
     glutIdleFunc(action);
     glutMouseFunc(mouse);
     glutMainLoop();
     delete game;
}
```

كود الملف الأساسي main.cpp هو كود رسومي استعملنا فيه دوال OpenGL, و كما سبق و ذكرت, شرح مفصل لكل هذا موجود في الدروس السابقة, ما يجب ملاحظته هو إستدعاء الدالة `game->play();` داخل الدالة `Display(void)`.

أما الكائن CGame فيتم إنشاؤه في الدالة الرئيسية main:

```
game = new CGame();
```

المشروع بسيط جدا و سنهتم بالأساسيات فقط, لن يتم إضافة خيارات للعبة, كإعادة اللعب و التراجع عن الحركة....الخ

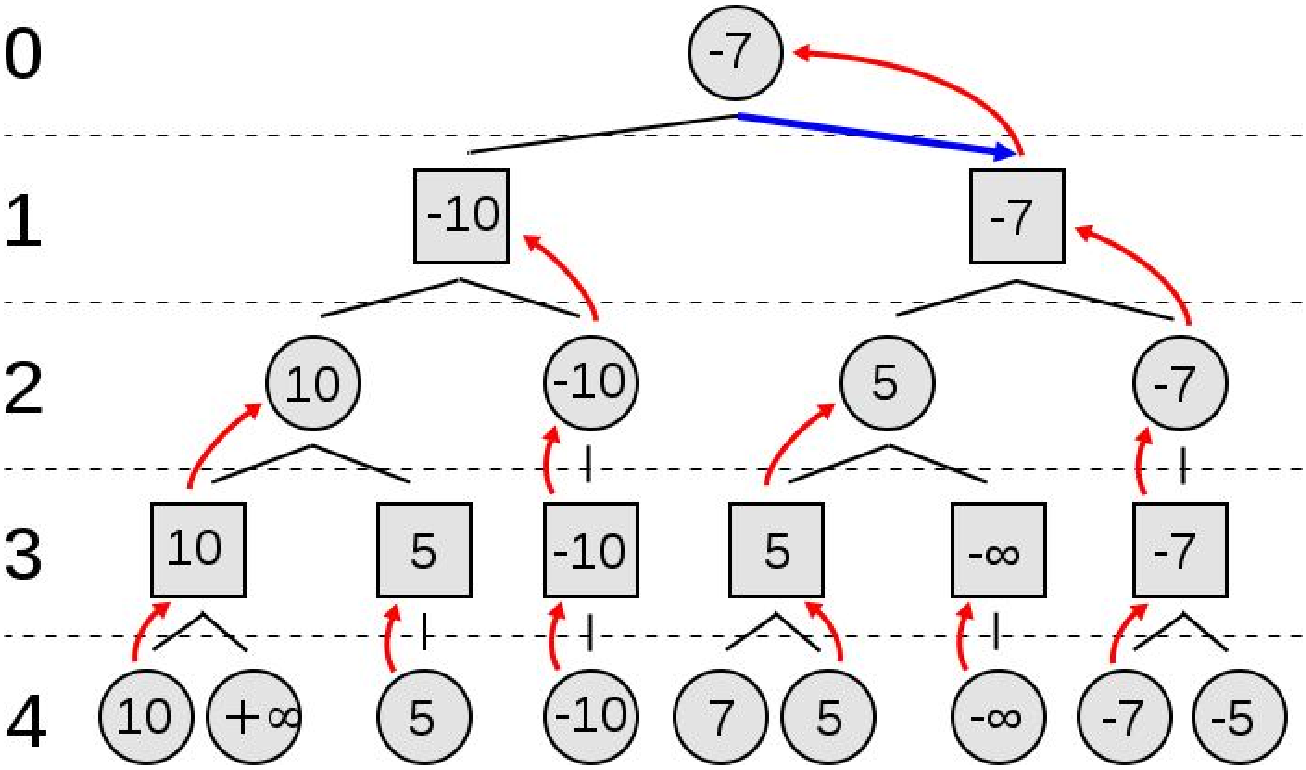
يمكنك تحميل المشروع من هذا الرابط:

http://www.mediafire.com/file/gb84g15v7r5fn70

2. الألغوريتم minimax

في كل دور يقوم بلعبه الحاسوب, يطبق فيه الألغوريتم minimax لإيجاد أفضل حركة ممكنة, و لإختيار أنسب الحركة, يولد البرنامج شجرة تمثل كل الحركات الممكنة مع نتيجة كل حركة و يختار الحركة الموالية ذات أفضل نتيجة.

مثال, نفترض أنه لدينا لعبة حيث كل لاعب له الخيار بين حركتين على الأكثر, و في هذا الدور سيقوم الحاسوب باللعب, أولا سيولد قائمة بكل الحركات الممكنة لخطوة واحدة, و من أجل كل حركة سيولد شجرة يحسب فيها كل الحركات التي يمكن للاعب المنافس أن يلعبها مع نتيجة كل حركة خلال أربعة أدوار فقط كما توضح الصورة:



كل مربع يمثل حركة يمكن أن يقوم بها الحاسوب و نتيجتها و كل دائرة تمثل حركة يقوم بها اللاعب و نتيجتها, إذا مهمت الحاسوب هي إختيار حركة بحيث يقلل أكبر خسارة ممكنة أو يفوز.

الألغوريتم يحسب النتائج بإستخدام دالة تقييم منفصلة تختلف حسب نوع اللعبة و قوانينها, القيمة ∞- تمثل الخسارة و القيمة ∞+ تمثل الفوز.

في الدور 3, من أجل كل مربع, يقوم الحاسوب باختيار الدائرة ذات أقل نتيجة ممكنة و يخزن نتيجتها في المربع, أي سيختار الحركة التي ستعطي أقل نتيجة للاعب, مثال, في المربع على أقصى اليسار سيختار الحاسوب بين الدائرتين 10 و ∞+, بما أن القيمة ∞+ تمثل الفوز, فلا يجب اختيارها لأنها ستعود إلى المربع و هو دور اللاعب, لذلك سيختار الحاسوب الحركات التي ستعطي أقل نتيجة للاعب, في هذه الحالة 10.

في الدور 2, الحاسوب سيختار الحركة ذات أكبر نتيجة ممكنة لأنها ستعود عليه, بين القيمة 5 و القيمة 10 سيختار 10.

بتطبيق هذه القواعد و تحليل كل الشجرة سنجد أن أفضل حركة سيختارها الحاسوب في هذه الحالة هي الحركة ذات النتيجة 7- على اليمين.

طبعا هذه الشجرة يتم تطبيقها على حركة واحدة ممكنة, نفترض أن الحاسوب سيلعب دوره على لعبة تيك تاك توو بحيث على اللوحة هناك 6 خانات فارغة, أي هناك 6 حركات ممكنة, من أجل كل حركة سيولد شجرة مثل السابقة و من أجل كل حركة في كل الأشجار المولدة سيتم توليد أشجار فرعية و البحث فيها عن أفضل حركة, أي أن الحاسوب أثناء البحث سيلعب كل الحركات الممكنة و يرى النتيجة و هذا سيأخذ وقت كبير و خاصة في الألعاب الأكثر تعقيدا, لذلك يتم إستعمال العمق, عمق الشجرة أو عدد مستويات البحث كما فعلنا في المثال.

بعد أن فهمنا مبدأ الألغوريتم, لنلقي نظرة على الألغوريتم.

سيكون لدينا ثلاث دوال, MinMax الدالة الرئيسية للألغوريتم, MaxMove الدالة التي ستعيد الحركة ذات أكبر نتيجة لصالح الحاسوب, MinMove الدالة التي ستعيد الحركة ذات أقل نتيجة للاعب.

```
MinMax (GamePosition game) {
  return MaxMove (game);
}
```

الدالة MinMax, من أجل كل حركة ممكنة game سنرجع الحركة ذات أكبر نتيجة بإستخدام الدالة MaxMove.

```
MaxMove (GamePosition game) {
  if (GameEnded(game)) {
    return EvalGameState(game);
  }
  else {
    best_move < - {};
    moves <- GenerateMoves(game);
    ForEach moves {
      playMove(game);
       move <- MinMove(game);
       if (Value(move) > Value(best_move)) {
          best_move < - move;
       }
     unPlayMove(game);
    }
    return best_move;
  }
}
```

في الدالة MaxMove سنرجع أفضل حركة, الحركة ذات أكبر نتيجة, لذلك نولد القائمة moves, قائمة كل الحركات الممكنة في الدور التالي و من إجل كل حركة ممكنة game, يلعب الحاسوب هذه الحركة عن طريق `playMove(game)` ثم يستدعي الدالة `MinMove(game)` من أجل الحركة game التي تم لعبها حتى ترجع لنا الحركة ذات أقل نتيجة ممكنة, و هي تمثل دور اللاعب, طبعا الألغوريتم تراجعي, أي أن كل دالة تستدعي الدالة الأخرى إلى أن تنتهي اللعبة و نحصل على فائز أو نصل إلى عمق محدد, و في الأخير نخزن في `best_move` الحركة ذات أكبر نتيجة لصالح الحاسوب, و نواصل إلى أن نحصل على أفضل حركة, نرجعها و ينتهي الأمر.

لكن قبل ذلك لا بد من تعريف الدالة MinMove, طبعا هي مشابهة للدالة السابقة غير أنها تقوم بالعكس.

```
MinMove (GamePosition game) {
  best_move <- {};
  moves <- GenerateMoves(game);
  ForEach moves {
     playMove(game);
     move <- MaxMove(game);
     if (Value(move) > Value(best_move)) {
        best_move < - move;
     }
     unPlayMove(game);
  }
  return best_move;
}
```

الألغوريتم السابق في أبسط أشكاله, نظري فقط, لذلك, عند برمجته سنخزن كل الحركات ذات أفضل نتيجة في قائمة و في كل مرة سنختار حركة عشوائية من هذه القائمة, طبعا الحركات تختلف لكن نتيجتها متساوية.

نعود إلى مشروعنا و نبرمج الألغوريتم, عد إلى الملف game.h و أضف له تعريفات الدوال و الثوابت التي سنحتاجها:

```
#include <list>

#define PI          3.14159265
#define INFINITY    1000000

typedef struct Move{
     int x,y;
};

class CGame
{
     .
     .
private:
      .
      .
      void generateMoves(std::list<Move> &moveList);
      int evaluatePosition(int forPlayer);
      void playMove(int x, int y, int player);
      void unPlayMove(int x, int y);
      Move MiniMax();
      int MinMove();
      int MaxMove();
      .
      .
```

الأسطر ذات الخلفية الرمادية هي ما تم إضافته للكود.

عرفنا عن الثابت INFINITY و عن التركيبة Move ذات المتغيرين x و y إحداثيات كل خانة من تسع خانات, أي أن قيمهم من 0 إلى 2.

سنشرح كل دالة أثناء كتابة كود الجسم, نضيف تعريف الدوال على الملف game.cpp

الدالة generateMoves دالة بسيطة و سهلة كل ما ستقوم به هو توليد كل الحركات الممكنة للحاسوب, في حالة مشروعنا, لعبة تيك تاك توو, الحركات الممكنة هي كل الخانات الفارغة التي يمكن أن يضع فيها الحاسوب الرمز O, سنجعل الحاسوب يلعب بالرمز O و اللاعب بالرمز X:

```
void CGame::generateMoves(std::list<Move> &moveList) {
     for(int i = 0; i < 3; i++){
          for(int j = 0; j < 3; j++)
                if(this->m_board[i][j] == EMPTY) {
                     Move m;
                     m.x = i; m.y = j;
                     moveList.push_front(m);
                }
     }
}
```

إذا هي عبارة عن حلقة تتحقق من كل الخانات و تقوم بإرداج الخانات الفارغة في القائمة moveList.

الدالة evaluatePosition تقيم حالة اللعبة بعد إجراء حركة ما, أي أنها سترجع إما +∞ في حالة فوز الحاسوب, طبعا اتفقنا على أن الحاسوب سيلعب بالرمز O, و ترجع -∞ إذا فاز اللاعب, أو XWIN, و في حالة التعادل ترجع 0, غير ذلك, أي في حالة أن اللعبة لم تنتهي بعد مع إجراء آخر حركة, ترجع الدالة -1.

```
int CGame::evaluatePosition(int forPlayer) {
     int state = this->checkGameState(forPlayer);
     if(state == XWIN || state == OWIN || state == DRAW) {
          if(state == OWIN) {
               return +INFINITY;
          } else if(state == XWIN) {
               return -INFINITY;
          } else if(state == DRAW) {
               return 0;
          }
     }
     return -1;
}
```

هذه اللعبة بسيطة جدا, لذلك فإن دالة تقييم الحركات بسيطة و سهلة, في ألعاب أخرى كالشطرنج ستكون هذه الدالة أكثر تعقيدا إذ ستعتمد في الحساب على وزن قطع الشطرنج.

الدالتين playMove و unPlayMove تقومان بإجراء حركة و بإلغائها على الترتيب, إثناء توليد شجرة الإحتمالات سيقوم الحاسوب بإجراء كل الحركات الممكنة على اللوح و سيحتاج إلى إلغاء الحركات لإجراء حركات أخرى.

```
void CGame::playMove(int x, int y, int player){
     this->m_board[x][y] = player;
}
void CGame::unPlayMove(int x, int y){
     this->m_board[x][y] = EMPTY;
}
```

نصل إلى الثلاث دوال الرئيسية, أولا الدالة MiniMax,

```
Move CGame::MiniMax() {
     int best_val = -INFINITY;
     std::list<Move>::iterator Iter;
     std::list<Move> moveList;
     std::list<Move> bestMoves;
     this->generateMoves(moveList);
     while(!moveList.empty()) {
          this->playMove(moveList.front().x, moveList.front().y,
0);
          int val = this->MinMove();
          if(val > best_val) {
               best_val = val;
               bestMoves.clear();
               Move m = moveList.front();
               bestMoves.push_front(m);
          }else if(val == best_val) {
               Move m = moveList.front();
               bestMoves.push_front(m);
          }
          this->unPlayMove(moveList.front().x, moveList.front().y);
          moveList.pop_front();
     }
     Iter = bestMoves.begin();
     int size = bestMoves.size();
     if(size > 1) {
          int temp = (rand() % size);
          for(int i=0; i<temp; i++)
               Iter++;
     }
     return *Iter;
}
```

المتغير المحلي best_val سنخزن فيه أفضل نتيجة سنصل إليها, نبدأ ب ∞- ثم كل ما نجد نتيجة أكبر نخزنها و نخزن الحركة الموافقة لكل أفضل نتيجة في القائمة bestMoves لأننا سنختار حركة عشوائية من القائمة ليلعبها الحاسوب.

القائمة الثانية هي moveList حيث سنولد كل الحركات الممكنة التالية و نخزنها فيها في انتظار أن يلعب الحاسوب كل حركة ممكنة و يرى النتيجة.

و باستخدام الحلقة `while(!moveList.empty())` نقوم بلعب كل الحركات الممكنة.

الحاسوب يستخدم الرمز O, لذلك استخدمنا الدالة:

```
this->playMove(moveList.front().x, moveList.front().y, O);
```

حيث نرسم O في أول خانة متاحة مخزنة على قائمة الحركات الممكنة.

و باستدعاء الدالة MinMove نحصل على القيمة val و هي نتيجة الحركة الحالية بعد تطبيق الدالة MinMove عليها, ثم نقارنها مع best_val و نأخذها كأحسن نتيجة في حالة كونها أكبر من best_val.

الدالة MinMove تكون كالتالي:

```
int CGame::MinMove() {
	int pos_value = this->evaluatePosition(O);
	if(pos_value != -1) {
		return pos_value;
	}
	int best_val = +INFINITY;
	std::list<Move> moveList;
	this->generateMoves(moveList);
	while(!moveList.empty()) {
	  this->playMove(moveList.front().x, moveList.front().y, X);
	  int val = this->MaxMove();
	  if(val < best_val) {
		  best_val = val;
	  }
	  this->unPlayMove(moveList.front().x, moveList.front().y);
	  moveList.pop_front();
	}
	return best_val;
}
```

أولا نتحقق هل حصلنا على فائز أم لا, في حالة استمرار اللعبة نواصل.

هذه الدالة تمثل دور اللاعب, أي سنرسم X على اللوح, و بعد لعب كل حركة X نتحقق هل `val < best_val` حتى نحصل على أصغر نتيجة يمكن أن يحصل عليها اللاعب لمصلحة الحاسوب طبعا.

و العكس في الدالة MaxMove, سنرسم الرمز O, أي دور الحاسوب, و سنرجع أفضل نتيجة يمكن أن يلعبها الحاسوب:

```
int CGame::MaxMove() {
     int pos_value = evaluatePosition(X);
     if(pos_value != -1) {
          return pos_value;
     }
     int best_val = -INFINITY;
     std::list<Move> moveList;
     this->generateMoves(moveList);
     while(!moveList.empty()) {
       this->playMove(moveList.front().x, moveList.front().y, O);
       int val = this->MinMove();
       if(val > best_val) {
               best_val = val;
       }
       this->unPlayMove(moveList.front().x, moveList.front().y);
       moveList.pop_front();
     }
     return best_val;
}
```

طبعا يجب أن تلاحظ أنه عند إستدعاء الدالة MaxMove فإن هذا الإستدعاء يمثل دور الحاسوب, نرسم O, نجري الحسابات ثم نستدعي الدالة MinMove, نرسم X و نجري الحسابات الضرورية و هكذا, و كأننا نمرر الدور من الحاسوب إلى اللاعب ثم من اللاعب إلى الحاسوب و في كل مرة إما نختار أفضل نتيجة أو أقل نتيجة حسب الدور إلى أن تنتهي اللعبة.

قبل أن ننهي المشروع, لا بد من تغيير شيء أساسي, و هو على مستوى الدالة play, سنغير الأسطر الي يتم فيها رسم O عن طريق كليك و نجعل الحاسوب يرسم O استنادا إلى الألغوريتم:

```
void CGame::play(void){
 if(this->m_state == PLAYING) {
     if(this->m_turn == O){
          Move move = MiniMax();
          this->m_board[move.x][move.y] = O;
          this->m_turn = X;
          this->m_state = this->checkGameState(O);
     }else if(this->m_clickedX != -1 && this->m_clickedY != -1){
...
```

هنا ننهي شرح هذا الألغوريتم, لتحميل المشروع بعد إضافة ما سبق إضغط هنا:

http://www.mediafire.com/file/287tfpja8ay9cji

3.الألغوريتم minimax 2

هو نفسه الألغوريتم السابق لكن سنجعل كل شيء يتم في الدالة ()minimax, سنلغي الدوال MaxMove و MinMove.

طبعا نفس المبدأ, فقط سنستعمل جملة شرطية لمعرفة نوع اللاعب, أذا كان O ننفذ الكود الخاص ب MaxMove و إذا كان X ننفذ كود MinMove, فقط لا غير.

سنضيف الدالة الجديدة و سنستعملها دون حذف الدوال السابقة, حتى نصل لمشروع فيه كل الدوال الممكنة و نستعمل في كل مرة دالة مختلفة و نرى الفرق من حيث سرعة التنفيذ و صغر الكود.

نعرف الدالة minimax2 داخل الملف game.h, و أيضا سنعرف تركيبة جديدة بإسم MS حيث سيكون فيها الحركة الممكنة و النتيجة معا.

```
typedef struct MS{
	int x,y;
	int score;
};

class CGame
{
...
private:
	...
	//minimax2
	MS *minimax2(int player);
	...
```

بما أن كل العمل سيتم في نفس الدالة, فستكون الدالة مزودة ببارامتر باسم Player حيث سيأخذ في كل إستدعاء للدالة إما القيمة X أو O حسب اللاعب الذي سيلعب هذا الدور.

جسم الدالة سيكون كالتالي, على مستوى الملف game.cpp:

```
MS *CGame::minimax2(int player){
	MS *ms = new MS();
	int pos_value = this->evaluatePosition(O);
	if(pos_value != -1) {
		ms->score = pos_value;
		return ms;
	}
	std::list<Move> moveList;
	int theOtherPlayer;
	player == X ? theOtherPlayer = O : theOtherPlayer = X;
	this->generateMoves(moveList);
	if (player == O) {
		ms->score = -INFINITY;
		while(!moveList.empty()) {
```

```
               this->playMove(moveList.front().x,
moveList.front().y, player);
               int score = minimax2(theOtherPlayer)->score;
               this->unPlayMove(moveList.front().x,
moveList.front().y);
               if (score > ms->score) {
                    ms->score = score;
                    ms->x = moveList.front().x;
                    ms->y = moveList.front().y;
               }
               moveList.pop_front();
          }
     }
     else {
          ms->score = +INFINITY;
          while(!moveList.empty()) {
            this->playMove(moveList.front().x, moveList.front().y,
player);
            int score = minimax2(theOtherPlayer)->score;
            this->unPlayMove(moveList.front().x,
moveList.front().y);
            if (score < ms->score) {
                 ms->score = score;
                 ms->x = moveList.front().x;
                 ms->y = moveList.front().y;
            }
            moveList.pop_front();
          }
     }
     return ms;
}
```

نفس العمل السابق المقسم على ثلاث دوال جمعناه في دالة واحدة, حيث كل قسم من الجملة الشرطية `if (player == O)` يمثل دورا للاعب, إما O أو X.
و في كل مرة نستدعي الدالة `minimax2` بشكل تراجعي يتم الإستدعاء من أجل اللاعب الآخر, حيث تكون القيمة في `theOtherPlayer`.

لتحميل المشروع بعد إضافة `minimax2` انقر على الرابط التالي:

http://www.mediafire.com/file/bcyvr8muexpuv4i/TicTacToe._minimax2.zip

4. الألغوريتم negamax

يعتمد الألغوريتم negamax على حقيقة أن max(a,b) = -min(-a,-b) لتبسيط الألغوريتم minimax, أي سنقوم بدمج دور كل من الدالتين MinMove و MaxMove في دالة واحدة.
المبدأ الأساسي هو: "ما هو في صالحي سيكون ضد مصلحة اللاعب الآخر".
بمعنى آخر, كل لاعب سيختار أكبر قيمة, ثم يرجعها مضروبة في -1.

```
int negamax(int depth)
{
      if (game_over or depth <= 0)
            return eval();
      int best_score = -INFINITY;
      move best_move;
      for (every possible move m) {
            play move m;
            int score = -negamax(depth - 1)
            unplay move m;
            if (score >= best_score){
                  best_score = score ;
                  best_move = m ;
            }
      }
      return best_score;
}
```

بسيطة, أفضل نتيجة للحركة الحالية هي أفضل نتيجة للحركة التالية للاعب الآخر مضروبة في -1.
سنضيف إلى المشروع دالتين, الدالة evaluatePosition2 و الدالة negamax.
على مستوى الملف game.h, أضف تصريحات الدوال:

```
class CGame
{
....
private:
      //negamax
      MS *negamax(int player);
      int evaluatePosition3();
```

نعرف كود الدوال على game.cpp, أولا الدالة evaluatePosition3:

```
int CGame::evaluatePosition2() {
 int state = this->checkGameState(O);
 if(state == OWIN)
  return -INFINITY;
 else if(state == DRAW)
  return 0;
 state = this->checkGameState(X);
 if(state == XWIN)
```

```
  return -INFINITY;
 return -1;
}
```

حسنا, بما أن كلا اللاعبين, الحاسوب و اللاعب, أصبحا يبحثان عن أكبر نتيجة (سابقا كان الحاسوب يبحث عن أكبر نتيجة و اللاعب عن أقل نتيجة لمصلحة الحاسوب) ثم يرجعها مضروبة في -1, فإن دالة التقييم ستتغير قليلا, عند الوصول إلى حركة ستمكن أحد اللاعبين من الفوز سنرجع النتيجة -∞ سواء للحاسوب أو اللاعب, لأنه إذا كانت الحركة الموالية نتيجتها -∞ (فوز أحد اللاعبين) و تم ارجاعها فسيتم تحويلها إلى +∞.

الدالة negamax, أضف الكود إلى game.cpp:

```
MS* CGame::negamax(int player)
{
	MS *ms = new MS();
	int pos_value = this->evaluatePosition2();
	if(pos_value != -1) {
		ms->score = pos_value;
		return ms;
	}
	ms->score = -INFINITY;
	std::list<Move> moveList;
	int theOtherPlayer;
	player == X ? theOtherPlayer = O : theOtherPlayer = X;
	this->generateMoves(moveList);
	while(!moveList.empty()) {
		this->playMove(moveList.front().x, moveList.front().y, 
player);
		int score = -negamax(theOtherPlayer)->score;
		this->unPlayMove(moveList.front().x, moveList.front().y);
		if (score >= ms->score) {
			ms->score = score;
			ms->x = moveList.front().x;
			ms->y = moveList.front().y;
		}
		moveList.pop_front();
	}
	return ms;
}
```

تطبيق حرفي لما تم ذكره في الألغوريتم فقط لا أكثر.

النتيجة score نحصل عليها من تنفيذ السطر:

```
int score = -negamax(theOtherPlayer)->score;
```

ثم نختار أكبر نتيجة عن طريق الجملة الشرطية `if (score >= ms->score)`.

في الأخير, نغير في الدالة play حتى نجعل الإختيار يتم عن طريق الدالة negamax:

```
void CGame::play(void){
	if(this->m_state == PLAYING) {
		if(this->m_turn == 0){
			//negamax
			MS *ms = negamax(0);
			this->m_board[ms->x][ms->y] = 0;
			delete ms;
```

لتحميل المشروع بعد كتابة الدالة negamax اضغط على الرابط التالي:

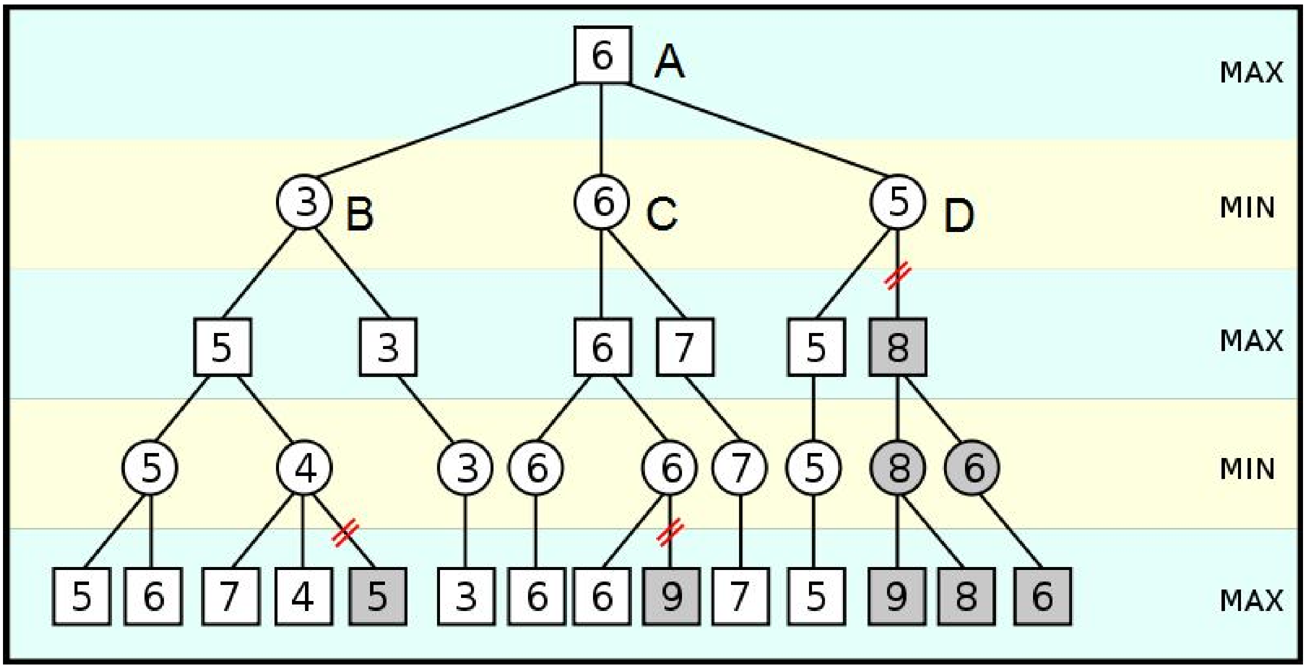
http://www.mediafire.com/?445hz19at5babzg

5.الألغوريتم minimax alpha-beta

في الحقيقة الألغوريتم السابق ليس فعالا جدا, في حالة الألعاب التي لديها مجموعة حركات كبيرة ستكون هناك مشكلة في استعمال الألغوريتم السابق, مثلا في لعبة الشطرنج قد يكون لدينا في مرحلة ما أكثر من 50 تحرك ممكن, باستعمال ما تعلمناه إلى حد الآن لن يتمكن الحايوب من تحليل أكثر من 4 أو 5 مستويات فقط خلال وقت معقول, أما تحليل كل الحركات الممكنة فسيأخذ وقتا كبيرا جدا.

لذلك, سنحتاج إلى طريقة لتسريع الألغوريتم السابق, سواء minimax أو negamax, إذا كان االاعب إنسان, فسيقول في حركة معينة: "إذا حركت الوزير إلى هناك سيتم أخذها من طرف حصان الخصم, لذلك لن أحركها هناك", أما إذا كان الدور للحاسوب فلن يفكر هكذا, بل سيدرس كل الإحتمالات الموالية لنفس الحركة و سيستغرق وقت كبير جدا ليجد في النهاية أن تلك الحركة ستسبب ضياع الوزير من طرف حصان اللاعب.

تقطيع ألفا-بيتا, Alpha-Beta pruning, سيستعمل المفهوم السابق لإزالة عدد معتبر من فروع أشجار البحث عن أفضل حركة,كيف سيفعل هذا, أثناء البحث, إذا ظهرت حركة ممكنة بحيث نتيجتها أقل من أحسن نتيجة تحصلنا عليها إلى حد الآن, لن يقوم الحاسوب بالبحث في بقية فروع تلك الحركة:



أنظر إلى الشجرة في الصورة, تصفح الشجرة يتم من اليسار إلى اليمين, لدينا المربع A يمثل الحاسوب(maximizer), سيختار طبعا أكبر قيمة من فروعه, و الدوائر B, C و D تمثل اللاعب و سيختار أقل نتيجة ممكنة, عندما نصل إلى الدائرة D (minimizer) و بعد تجول الحاسوب في فرعها الأيسر, حصل على النتيجة 5, و بما أن Dسيختار أقل نتيجة فإن أكبر نتيجة يمكن أن يختارها هي 5, و في

نفس الوقت لدينا قيمة C هي 6, و A هو mximizer أي سيختار أكبر قيمة, أي أنه قطعا لن يختار القيمة التي سترجع من D مهما كانت لأنها ستكون أقل أو تساوي 5 و هي أقل من 6, لذلك لا داعي لأن يبحث الحاسوب عن نتائج الفرع الأيمن للدائرة D و سيتم تجاهله.

الألغوريتم minimax_alphabeta:

```
int alphabêta(int depth, int alpha, int bêta)
{
   if (game over or depth <= 0)
      return winning score or eval();
   move bestMove;
   if(nœud == MAX) { // computer's turn
      for (each possible move m) {
         make move m;
         int score = alphabêta(depth - 1, alpha, bêta)
         unmake move m;
         if (score > alpha) {
            alpha = score;
            bestMove = m ;
            if (alpha >= bêta)
               break; /* Beta cut-off */
         }
      }
      return alpha ;
   }
   else { //player's turn
      for (each possible move m) {
         make move m;
         int score = alphabêta(depth - 1, alpha, bêta)
         unmake move m;
         if (score < bêta) {
            bêta = score;
            bestMove = m ;
            if (alpha >= bêta)
               break; /* Alpha cut-off */
         }
      }
      return bêta;
   }
}
```

لنكتب كود هذا الألغوريتم, سنضيف تعريف دالة تقييم جديدة و دالة الألغوريتم, على مستوى الملف game.h:

```
class CGame
{
...
private:
     ...
     //minimax alphabeta
     int evaluatePosition3();
     MS *minimax_alphabeta(int player, int alpha, int beta);
```

إلى الملف game.cpp حيث سنكتب أجسام الدوال, دالة التقييم evaluatePosition3 كسابقاتها مع تعديل بسيط:

```
int CGame::evaluatePosition3() {
	int state = this->checkGameState(O);
	if(state == OWIN)
		return +INFINITY;
	if(state == DRAW)
		return 0;
	state = this->checkGameState(X);
	if(state == XWIN)
		return -INFINITY;
	return -1;
}
```

ترجع ∞+ إذا فاز الحاسوب و ∞- إذا فاز اللاعب و صفر في التعادل و 1- إذا كانت اللعبة مستمرة.
سنطبق ما تكلمنا عنه عن تقطيعات ألفا بيتا على مستوى الدالة minimax_alphabeta:

```
MS* CGame::minimax_alphabeta(int player, int alpha, int beta)
{
 MS *ms = new MS();
 int pos_value = this->evaluatePosition3();
 if(pos_value != -1) {
		ms->score = pos_value;
		return ms;
 }
 std::list<Move> moveList;
 int theOtherPlayer;
 player == X ? theOtherPlayer = O : theOtherPlayer = X;
 this->generateMoves(moveList);
 if (player == O) {
  ms->score = -INFINITY;
  while(!moveList.empty()){
   this->playMove(moveList.front().x, moveList.front().y, player);
   int score = minimax_alphabeta(theOtherPlayer, alpha, beta)-
>score;
   this->unPlayMove(moveList.front().x, moveList.front().y);
   if(score > alpha) {
    alpha = score;
     ms->x = moveList.front().x;
     ms->y = moveList.front().y;
     if(alpha >= beta) break;
   }
   moveList.pop_front();
  }
  ms->score = alpha;
 }
 else {
  ms->score = +INFINITY;
  while(!moveList.empty()) {
   this->playMove(moveList.front().x, moveList.front().y, player);
   int score = minimax_alphabeta(theOtherPlayer, alpha, beta)-
>score;
   this->unPlayMove(moveList.front().x, moveList.front().y);
```

```
    if (score < beta) {
      beta = score;
      ms->x = moveList.front().x;
      ms->y = moveList.front().y;
      if(alpha >= beta) break;
    }
    moveList.pop_front();
   }
  ms->score = beta;
 }
 return ms;
}
```

و بتغيير بسيط في الدالة play سنتمكن من إستعمال هذه الدالة:

```
void CGame::play(void){
     if(this->m_state == PLAYING) {
          if(this->m_turn == 0){
               //minimax alphabeta
               MS *ms = minimax_alphabeta(0, -INFINITY, +INFINITY);
               this->m_board[ms->x][ms->y] = 0;
               delete ms;
```

لتحميل المشروع بعد إضافة الدالة minimax_alphabeta:

http://www.mediafire.com/?6fmi45vczi9eu03

6. الألغوريتم negamax alpha-beta

و كخطوة أخيرة في هذا الدرس, سنجمع بين negamax و تقطيعات alpha-beta حسب الألغوريتم التالي:

```
int ALPHA_BETA(depth, Alpha, Beta)
{
 if(game_over or depth <= 0)
     return eval();
 best_score = -INFINITY;
 move best_move;
 for (every possible move m) {
 play move m;
 int score = -ALPHA_BETA(depth-1,-Beta,-Alpha);
 unplay move m;
 if (score >= best_score){
  best_score = score;
  best_move = m;
  if (score > alpha){
   Alpha = score;
   if(Alpha >= Beta) break;    //cut-off
  }
```

```
 }
 return best_score;
}
```

لتطبيق الألغوريتم سنضيف الدالة negamax_alphabeta إلى المشروع, أولا نصرح عنها في game.h:

```
class CGame
{
...
	//negamax alphabeta
	MS *negamax_alphabeta(int player, int alpha, int beta);
```

و في الملف game.cpp سنكتب كود جسم الدالة:

```
MS* CGame::negamax_alphabeta(int player, int alpha, int beta)
{
	MS *ms = new MS();
	int pos_value = this->evaluatePosition2();
	if(pos_value != -1){
		ms->score = pos_value;
		return ms;
	}
	ms->score = -INFINITY;
	std::list<Move> moveList;
	int theOtherPlayer;
	player == X ? theOtherPlayer = O : theOtherPlayer = X;
	this->generateMoves(moveList);
	while(!moveList.empty()){
		this->playMove(moveList.front().x, moveList.front().y, 
player);
		int score = -negamax_alphabeta(theOtherPlayer, -beta, -
alpha)->score;
		this->unPlayMove(moveList.front().x, moveList.front().y);
		if(score > ms->score){
			ms->score = score;
			ms->x = moveList.front().x;
			ms->y = moveList.front().y;
			if (score > alpha){
				alpha = score;
				if (alpha >= beta)
					break;
			}
		}
		moveList.pop_front();
	}
	ms->score = alpha;
	return ms;
}
```

ثم نجعل الدالة play تستعمل الدالة negamax_alphabeta لتوليد حركة الحاسوب:

```
void CGame::play(void){
      if(this->m_state == PLAYING) {
            if(this->m_turn == O){
                  //negamax alphabeta
                  MS *ms = negamax_alphabeta(O, -INFINITY, +INFINITY);
                  this->m_board[ms->x][ms->y] = O;
                  delete ms;
```

لتحميل المشروع النهائي بكل الدوال:

http://www.mediafire.com/?den8lfc2xr3imgb
أو
http://www.4shared.com/folder/ilQQUXIv/minimax.html

لا بد من ذكر أننا لم نستعمل الحد الأقصى للعمق المسموح للحاسوب تحليله, لأن اللعبة بسيطة و لن يستغرق الحاسوب وقتا طويلا في تحليل كل المستويات و كل الحركات الممكنة, لكن في لعبة أكثر تعقيدا مثل الشطرنج يجب وضع حد أقصى للعمق (depth), يتم التحقق في أول الدالة negamax_alphabeta أو أي دال ثانية تم اختيارها.

أعتذر عن كل الأخطاء الموجودة في الدرس و إن لم أتعمدها و لم ألاحظها و شكرا لكل كتاب المقالات التي اعتمدت عليها كمراجع في كتابة الدرس.

انتهى.